LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-659097

Exploring Distributed Memory Parallel CPLEX

G. Cong, J. Magerlein, D. Rajan, C. Meyers

August 21, 2014

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Exploring Distributed Memory Parallel CPLEX

Authors:

Guojing Cong (gcong@us.ibm.com) and John Magerlein (mager@us.ibm.com)

IBM T. J. Watson Research Center

Deepak Rajan (rajan3@llnl.gov) and Carol Meyers (meyers14@llnl.gov)

Lawrence Livermore National Laboratory

Nov. 30th, 2012

1 Summary

We evaluate the potential impact of distributed memory parallelization of CPLEX for solving unit commitment (UC) problems. Earlier studies have shown that mixed-integer programming (MIP) solvers behave quite differently from scientific codes (e.g., fluid dynamics or N-body simulation). Increasing the number of threads increases the number tree nodes explored per unit time (consider it as throughput) but does not necessarily improve the solution time as the search path can be different. Given more processors, effective use of parallelism to improve performance (to be exact, solution time) is a challenge.

In this work, the Watson high-performance application team worked with the CPLEX team for the implementation of two different distributed-memory solvers in MPI based on CPLEX. Each solver corresponds to one of the proposed parallelization strategies in earlier studies. With concrete implementations we are able to evaluate the pros and cons of each strategy.

The first strategy, denoted as approach *a* in this document, parallelizes the solve by utilizing concurrent CPLEX solvers with different parameter sets. In our study we evaluated more than twenty parameter sets (strategies), and show that with proper configurations, coordinated, concurrent search improves the solution time for all UC models compared with the original CPLEX using the same number of threads. In some cases the improvement is more than 50%. Although it is hard to predict the speedup with a massively parallel computer, this result is very promising considering how elusive it has been to achieve solid speedups.

The second strategy, denoted as approach *b*, parallelizes the branch-and-cut search on a distributed-memory cluster. Although direct distributed-memory parallelization of the branch-and-cut search process has reasonable parallel efficiency, it does not improve the solution time for the UC models due to frequent load balancing and interrupt to CPLEX in the early search phase. Here with an MPI implementation we are able to evaluate engineering choices such as communication mode and load-balancing frequencies.

Our study shows that the ultimate large-scale solver should be a combination of the two approaches, and can potentially bring significant performance improvement to solving these models.

2 Foundations for distributed-memory implementation

Approaches *a* and *b* are identified by the Livermore and IBM teams as two candidate strategies to speed up solving unit commitment problems

with CPLEX on massively parallel supercomputers. With approach *b*, the computing resources in a distributed memory machine (e.g., a cluster of SMPs) are used to explore the branch-and-cut search tree. The search tree will be distributed among computer nodes, and a mechanism needs to be developed to control assigning nodes to solvers, querying status, and exchanging information such as bounds and feasible solutions. As common in distributed tree search algorithms, synchronization and load-balancing among solvers are necessary. Specific to integration with CPLEX, migration of tree nodes among solvers in the load-balancing strategy has critical impact on performance. In contrast to approach *b*, the solvers in approach *a* are largely independent. It can be viewed as multiple search processes from different starting points in the search space. Minimal exchange of information is necessary. The challenge, however, is how these solvers can contribute collectively to the search.

Both approaches require a communication mechanism among machines (processes), which may be implemented with *socket*, *unix pipes*, and in the high performance computing (HPC) setting, *MPI*. There are some common functionalities found in both approaches, for example, starting a remote solve (from the root or from any tree node) and waiting for the remote solve to finish. The CPLEX *remote object* abstracts the interfaces needed in both approaches *a* and *b* (and other possible scenarios) to simplify the development of distributed algorithms. For example, a user can invoke a remote user function on the solvers from the master. The *remote object* handles automatically the communication between the two processes. In most cases, there is no need for the user to explicitly invoke the MPI communication primitives.

Up to three components comprise a distributed application in the current design. They include a *master* process, a group of solvers/workers (e.g., CPLEX on a shared-memory node), and an optional piece of dynamic library loaded into worker processes. This design is different from the prevalent SPMD paradigm in current HPC applications. To use the MPI communication mechanism, the MPI implementation on the target platforms needs to support the MPMD paradigm. The advantage is flexible deployment to a wide range of configurations.

In our evaluation and implementation of approaches *a* and *b*, the master manages the search strategies and the solvers. It calls CPLEX libraries for communicating with and controlling the workers. In this sense it can be viewed a client in the client/server model. Typically the communication is between the master and the workers but not among the workers themselves. This again is very different from most MPI programs in scientific computing. The master in approach *b* is fairly complex as it is responsible for managing the distributed search tree. It decides which nodes to send to remote solvers and how synchronization and load balancing are done. The master in approach *a* collects information from

the solvers and relays it to the group. In the simplest implementation, the master in approach *a* may simply start remotes solves and wait for their termination. Of course enhancement is necessary for fast solutions of the UC problems (see Section 4).

The workers are CPLEX processes invoked with the '–worker' option. After startup, they wait for messages/commands from the master. The differences between approach *a* and approach *b* are solely in the master and the dynamic libraries loaded in the worker.

The following shows how the components of a distributed solver are deployed with MPI (that supports MPMD) on a cluster.

The solve is invoked from command line as:

```
/scratch/gcong/openmpi-1.6/bin/mpirun  --app apppickedfile
```

apppickedfile specifies the layout of the processes, and an example is shown as follows.

```
-np 1  -machinefile hostfile2  ./rmt-mpi-parmipopt -output-prefixed \
       -model=/scratch/gcong/CPLEX/llnl/M07.rew
-np 1  -machinefile hostfile2 \
../../../bin/power64_sles10_4.1/static_pic/cplex -worker=mpi \
     -userfunction=mpi-parmipopt_userfunction=REGISTER_USERFUNCTION \
     -logfile=server0.log
-np 1  -machinefile hostfile1 \
     ../../../bin/power64_sles10_4.1/static_pic/cplex -worker=mpi \
     -userfunction=mpi-parmipopt_userfunction=REGISTER_USERFUNCTION \
     -logfile=server2.log
```

In this example, we start a master process (*rmt-mpi-parmipopt*) on *host2* that takes an input model *M07.rew*. Two workers are started on *host1* and *host2*.

In our study we use a two-node cluster. Each node has 8 IBM Power7 cores running at 3.61GHz, with each core capable of four-way simultaneous multithreading. Power7 executes instructions out-of-order. There are 12 execution units (including 2 fixed-point units and 2 load/store units) per core shared by the 4 hardware threads. Each core has 32KB L1, 256KB L2, and 4MB L3 caches.

In this document, we call the distributed-memory solver in approach *a* the *concurrent solver* and the solver in approach *b* the *parallel solver*. We refer to CPLEX on a shared-memory node as CPLEX or standalone CPLEX.

We used the following models. The performance and behavior of CPLEX on an SMP has been studied earlier.

```
MODEL ( M02 HiLoad ProdCost 0614 CPLEX ) 24 (ST step 24 sample 1MIP).rew
```

```
MODEL ( M03 HiLoad ProdCost 0614 CPLEX ) 22 (ST step 22 sample 1MIP).rew
MODEL ( M04 HiLoad ProdCost 0614 CPLEX ) 11 (ST step 11 sample 1MIP).rew
MODEL ( M05 HiLoad ProdCost 0614 CPLEX ) 18 (ST step 18 sample 1MIP).rew
MODEL ( M06 HiLoad ProdCost 0614 CPLEX ) 7 (ST step 7 sample 1MIP).rew
MODEL ( M07 AllGas ProdCost 0608 CPLEX ) 24 (ST step 24 sample 1MIP).rew
MODEL ( M09 AllGas Need 0608 ) 28 (ST step 28 sample 1 MIP).rew
MODEL ( M10 AllGas Need 0608 ) 19 (ST step 19 sample 1 MIP).rew
```

For simplicity, we denote these model files as M02.rew through M10.rew, respectively. Among these models, six (M02-M07) are of the slow-solving variety (they time out at the California ISO's time limit of 3 hours) under the 0.05% optimality tolerance that the customer requires.

3 Concurrent solves (approach a)

Parameter sets can have significant impact on the performance of CPLEX for the given models. It can take a long time (e.g., more than 1 hour and up to two hours) to solve most models on a single node. They are good candidates for trying concurrent solves. Earlier studies have identified two factors that make concurrent solve with different parameter sets crucial. First, we observe CPLEX spends a significant amount of time to process the root node. Second, concurrent solves can provide enough parallelism to populate a massively parallel computer (e.g., the BlueGene Q computer). In our study we first evaluate the efficiency of concurrent solves.

3.1 Parmipopt with *primal* and *dual*

The first implementation evaluated is *parmipopt*. In *parmipopt*, the master starts a few workers to work on the problem and waits for their termination. Regularly through the *remote object* mechanism each worker reports its best *primal* and *dual* bounds to the master. The master terminates the search when a predefined optimality gap is reached. *parmipopt* demonstrates how to use the CPLEX remote object to implement distributed solvers.

We first use two parameter sets on two MPI processes. The first parameter set cranks up heuristic parameters and disables cuts to focus mainly on the *primal* part of the problem. The second parameter set focuses on the *dual* part of the problem to improve the dual bound. By assigning different strategies to the processes, we expect upon termination the best primal and best dual bound come from different solvers.

Table 1 shows the solver that provided the best primal bound and dual bound for the UC models. Indeed we see the best primals come from the *primal* solver and the best duals come from the *dual* solver.

This demonstrates how we can use independent solvers in a distributed environment to speed up the search. To utilize more processors, the user can set up different random seeds for the *primal* and *dual* solvers by setting the *CPX_PARAM_RANDOMSEED* parameter that influences some of the heuristic choices made by CPLEX during the search.

Models	primal-index	dual-index
M02	0	1
M03	0	1
M04	0	1
M05	0	1
M06	0	1
M07	0	1
M09	0	1
M010	0	1

Table 1: Solvers that provide the best primal and dual in parmipopt. 0 represents the *primal* and 1 represents the *dual*

3.2 Augmenting parmipopt with more parameter sets

CPLEX has a large collection of parameters that emphasize different strategies. Instead of using simply *primal* and *dual*, we next consider running concurrently parmipopt with four parameter sets, that is, *primal*, *dual*, and *default* (the default set of CPLEX), and *no-flow-cut* (flow cuts are turned off. This was found effective by the Watson team on some other unit commitment problems). We include *default* to compare the performance of other sets against the default CPLEX set.

Table 2 gives the solvers that achieved the best primal bound and dual bound for the UC models. We see that all of them are from *default* (Here 0, 1, 2, 3 represent *primal*, *dual*, *default*, and *no-flow-cut*, respectively). On one hand, this test shows that the default parameter set of CPLEX is indeed a good fit for many problems; on the other hand, it shows the simple approach of completely independent searches probably does not effectively utilize the computing resources in the parallel computer.

We increased the number of processes (thus the number of heuristic sets) in our experiments from four to sixteen and then to twenty one. Each process adopts a different set of parameters. The helpful ones are listed in the appendix [1]. Occasionally we did observe the best bounds were provided by non-default solvers. For example, better performance with other sets than *default* were observed on M09 and M10. As these two

[1]Some of the parameters listed are subject to change.

Models	primal-index	dual-index
M02	2	2
M03	2	2
M04	2	2
M05	2	2
M06	2	2
M07	2	2
M09	2	2
M010	2	2

Table 2: The solvers that provide the best primal and dual with 4 independent solvers. *default* has index 2

models are easy to solve, we conclude that simple, concurrent searches are not very helpful.

4 Coordinated, concurrent solves

Inspecting the log files from the simple concurrent solver, it is clear to us that although the overall time spent is the least with *default*, during the life time at any certain moment other solvers do discover better solutions. In concurrent searches some solvers may by chance find good solutions before others. Instead of kept private, they can be shared with other solvers to help prune their search trees.

The coordination is currently implemented as follows. The master installs an *infocallback* and a *heuristiccallback* at the workers. Through the *infocallback* a solver reports its status to the master at regular intervals. The best primal and dual bounds together with the current best integer solution are sent to the master. The master keeps track of the current best primal bound and dual bound and the best feasible solution reported from all workers. Whenever a new solution is discovered, the master checks whether it is better than the current best, if so, it broadcasts the new bound and solution to all workers. At the workers, the *heuristiccallback* is called at every viable node in the branch-and-cut tree with the solution vector for the current relaxation as input. In this function we retrieve the incumbent and checks whether a solution with better objective has been received. If so, the *heursiticcallback* returns the new objective and received solution to CPLEX.

4.1 Behavior and performance

Table 3 shows with various sets the solvers that returned the final best primal and dual bounds. Note that in all runs each process uses a differ-

ent parameter set.

	16× 4		4× 8		8 × 16		8 × 8	
Model	dl	prml	dl	prml	dl	prml	dl	prml
M010	0	3	1	3	0	3	0	5
M02	2	2	0	3	2	2	2	2
M03	7	2	3	2	2	2	3	2
M04	7	9	2	3	2	2	0	6
M05	5	12	2	2	5	7	7	3
M06	10	7	2	2	7	7	6	3
M07	5	6	0	1	1	6	1	6
M09	0	7	1	3	2	6	1	6

Table 3: Solvers that provide the best primal and dual bounds

In table 3 the best primal and dual bounds are no longer always returned by the *default* solver. Other solvers provide the best bounds as well. Table 3 shows parameter sets 0, 1, 2, 3, 6, and 7 are suitable for the models. They are *aggressive root cuts*, *aggressive probe*, *no cuts*, *default*, *more gomory cuts*, *more probes*, respectively (for reference check the the detailed list of parameters in the appendix).

Table 4 shows the execution times with 16 processes, 4 threads per each process, and 8 processes, 8 threads per each process, in comparison with standalone CPLEX with 64 threads. For models M02, M03, M04, M05, and M06, the performance improves significantly. For M09 and M010, our distributed solver is slower but far below the 2-hour time limit. The performance for M07, however, becomes worse (in comparison with standalone CPLEX).

Coordinated, concurrent search with 64 threads

Models	16procs × 4thrs	8procs × 8thrs	SMP 64 thrs
M02	1174.54	1210.41	2309.50
M03	2694.65	1282.78	6376.42
M04	2827.35	3135.17	7531.00
M05	3493.98	1876.93	5157.15
M06	2163.56	1632.75	2323.75
M07	27334.9	33764.55	3278.63
M09	67.19	66.85	34.31
M010	71.19	53.80	33.98

Table 4: Performance of different settings

We run more experiments with different strategies in the hope to find

better parameter sets. Table 5 shows the performance of 4 processes each with 8 threads, 8 processes each with 16 threads, 32 processes each with 4 threads, and 4 processes each with 32 threads, for the UC models. Again for models M02, M03, M04, M05, M06 all parameter sets for the distributed solver perform well (finish within 1 hour). However, none of them find a satisfactory solution for M07 within 2 hours.

Experiments with more parameter sets

Models	4× 8 thrs	8 × 16	32 × 4	4 × 32
M02	1520.47	1772.02	1706.11	1521.34
M03	2284.45	2733.67	2333.94	2522.40
M04	1290.76	3093.91	3567.90	2673.96
M05	2357.25	3303.70	3038.62	3793.72
M06	1596.44	1785.52	2487.28	2965.64
M07	20389.71	57913.56	7200*	7200*
M09	63.70	82.67	85.68	67.55
M010	58.26	62.9	95.66	59.48

Table 5: More performance results of coordinated, concurrent solver. The entry with * timed out after 2 hours

4.2 Flexible configurations

Installing a control call back (such as the *heuristiccallback*) disables some MIP features in CPLEX. In our coordinated, concurrent-search solver, the default dynamic search is replaced with regular branch-and-cut, and that can be one of the reasons why our solve does not perform well on M07. As we have shown that the default set and strategy employed in CPLEX are in general effective and robust for UC problems, when there are enough processors, we consider keeping one instance of the *default* solver without the interference of control callbacks.

Ideally, the best primal bound among all current solvers should still be used to set the *cutoff* for the *default* solver, the current CPLEX implementation does not yet support dynamically changing the *cutoff* value. In our current implementation, we allow the user to control whether to instrument the *default* solver with the *heurisitccallback* during concurrent searches. One advantage of such configuration is that even in the worst-case, distributed search is at least as fast as standalone CPLEX.

Fig. 1 shows for the UC models the performance of the coordinated, concurrent solver against that of the standalone CPLEX using the same number of threads. For all models the concurrent solver is faster than CPLEX. Significant performance improvement is observed for M03, M04, M05, M07. Also all solves complete within 1 hour.

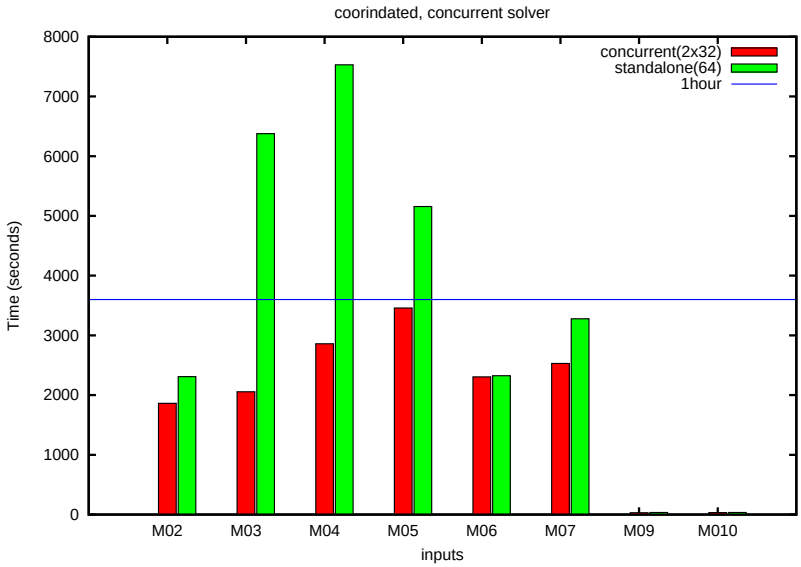

Figure 1: Performance comparison. In *concurrent*, one solver is CPLEX default, and the other uses the *more heuristic* strategy (see appendix for detailed parameter set)

We can imagine other configuration patterns for the concurrent solver. For example, the solvers do not have to use the same number of threads; callbacks are installed to a subset of the solvers; and information sharing occurs only among certain subsets. As for which configuration is the best for the UC models, machine learning techniques might be applied to identify some of the key features.

5 Parallel search (approach b)

Branch-and-bound has been well studied in the literature, yet the distributed implementation coupled with CPLEX for fast solution of unit commitment problems presents many questions and brings major engineering challenges.

The first question is whether there is enough parallelism, i.e., number of nodes in the tree, for a massively parallel computer. Earlier, we showed that for the UC models there *oftentimes* are many nodes waiting to be explored in the tree, thus there is potential performance reward in adopting a massive parallel solver.

The next question is when parallel search should be applied. Some of the CPLEX MIP features do not migrate easily to the distributed-memory setting. As the search space is enormous for any brute force branch-and-bound even on massively parallel computers, distributed branch-

and-bound starting from the root is not likely to perform well. We have shown that coordinated, concurrent searches are quite effective for most of the UC models. Parallel search should probably be combined with concurrent search, and start when all solvers have gathered enough nodes to explore and no significant reduction in the optimality gap has been made for a while.

The third question to answer is how many parallel processors will compensate the parallel overhead and bring tangible performance improvement for distributed branch-and-bound. Early studies revealed the limited increase in parallelism does not always improve the performance. This is largely due to the nature of the branch-and-cut search. With any strategy there is always the possibility that with more processors the search takes a tortuous path. Although the absolute number of nodes explored per unit time increases with the number of processors, the execution time may actually increase. At extreme scale for limited problem size, however, different behavior might be observed.

An implementation of parallel search is essential for answering some of these questions and evaluating design choices for the ultimate massive-scale solver. To this end, we first evaluate the PICO parallel solver. Unfortunately PICO+CPLEX ran into many problems at the time we tried, and we had to abandon that choice. Instead, we develop our own (prototype) implementation in MPI.

For a high-performance MPI implementation, there are many design choices, including how the tree is maintained (should there be a master centrally managing the tree or should there be a peer-to-peer system among the workers), whether the poll-and-push communication model or interrupt-and-pull model should be used, how frequent polling should occur, how many nodes stay within the CPLEX solver and how many nodes are shared in the distributed tree, how many nodes to grab during the load-balancing phase when a server runs out of work, and what data structure should be used for the distributed tree.

We make the following choices in our MPI implementation. We adopt the poll-and-push strategy between the master and the workers. That is, the master polls and pushes work to the workers at predefined intervals. As the number of nodes increase at each worker, the interval increases to reduce the polling overhead. There is one node in the distributed branch-and-cut tree representing a subtree in the solver. The node contains the set of bound changes and added local cuts relative to the root node. The distributed tree is stored in the master, and is implemented with a heap (using the associated lower bound as the key). When a worker runs out of work, first the nodes in the distributed tree are taken, then the master grabs nodes from other worker's internal tree. With the current design there is only one master managing the distributed search tree.

Fig. 2 shows the performance scaling of parallel search with a test

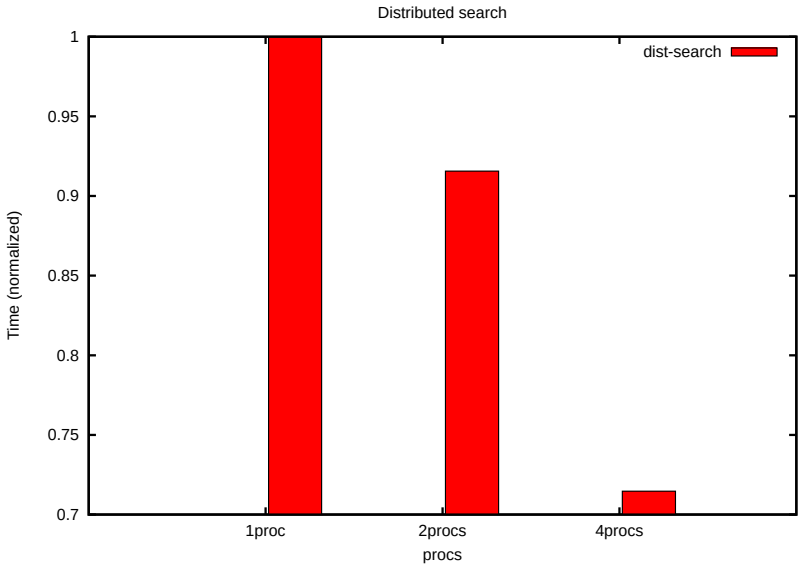

Figure 2: Distributed search performance

model with 1 MPI process, 2 MPI processes, and 4 MPI processes. As the number of processes increases, the execution time decreases. Although the parallel efficiency of parallel search is reasonable, the performance of parallel search on the UC models is far worse than coordinated concurrent searches, and even worse than standalone CPLEX. In fact, parallel search does not complete within the 2 hour time limit for M02, M03, M04, M05, M06, and M07. We had to abort the runs for these models. For example, parallel search did not finish within 5 hours for M02.

Several factors contribute to the poor performance of parallel search. The first is that frequent transferring of nodes during the initial phase of parallel search precludes certain optimization heuristics in CPLEX. The second is due to the interruption of the workers with constant polling. Also once each worker has enough nodes, there is no exchange of current best nodes among the solvers and some solvers can get stuck with a region with poor solutions. This behavior is reflected in the big difference between the current best solutions found by each solver during the search process. In our experiment with M02 for instance, one solver finds a solution with 6.39% gap, and for more than 60 time steps, the other solver is still stuck with the 99.31% gap.

Although straightforward parallel search (with current design choices) does not perform well on the current models, it actually complements the coordinated, concurrent search approach. In fact, we believe the ultimate distributed, massive-scale solver should be a combination of the two.

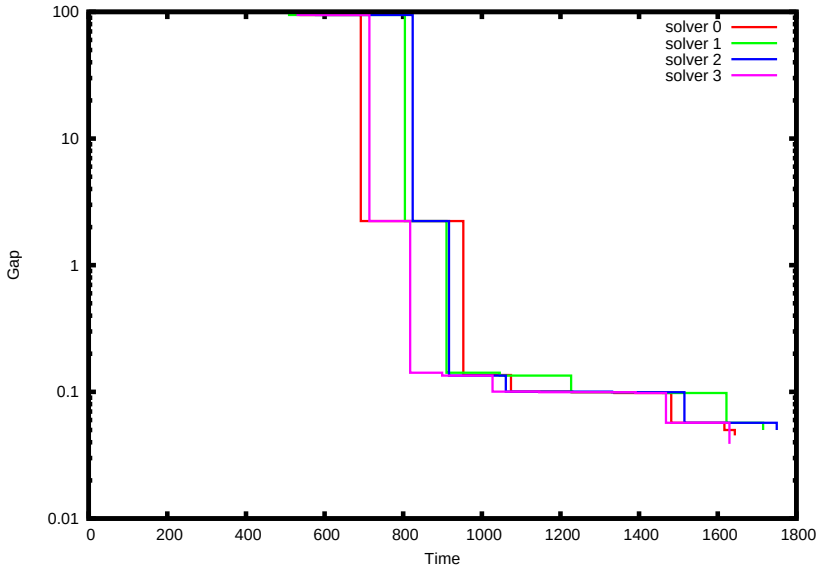

Figure 3: M02

6 Prospect of combining the two approaches

In section 3 we showed that the prompt exchange of bounds and solutions among concurrent solvers work well for most of the UC models. Intuitively, this approach can be considered as starting multiple different search heuristics in the neighborhood of the current good solutions. The advantage is that all solvers quickly catch up with each other and can collectively explore this neighborhood. Fig. 3 shows this lock step behavior of the concurrent solvers. All the step curves in the plot have similar shapes. And once there is a decrease of the optimality gap in one of the solvers, the others quickly follow.

Similar behavior is observed with other models. Fig. 4 shows the gap decrease step curve for M07. Again we see lock step decrease of gaps among the solvers. The problem with M07, however, is that for a very long period of time (between 1000 seconds and 3500 seconds), no solver is able to improve the gap. One can imagine a scenario where all solvers end up in the same neighborhood of the search space, and the work they do may be repetitive.

In section 5 we showed that parallel search at the top of the search tree can disrupt the search strategies of CPLEX with polls, stops, and grabbing nodes, and good solutions are not propagated quickly to all solvers. Yet parallel search is efficient in evaluating a large number of nodes when load balancing occurs rarely and there is minimal interrupt to CPLEX.

We propose combining concurrent solve with parallel solve based on

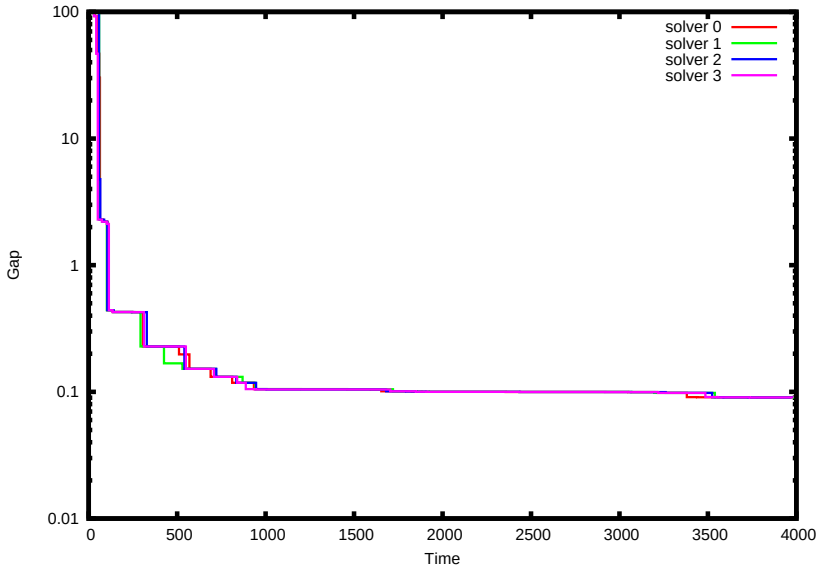

Figure 4: M07

our study. First, coordinated, concurrent search is invoked with as many different parameter sets as possible. It keeps running as long as the bounds improve (with large enough increments). When a large number of nodes are created on each process and the bounds stop improving for some iterations, we switch to parallel solve. At this stage, the distributed branch-and-bound happens down the search tree, and minimal or no load-balancing is necessary (thus minimal interrupt to CPLEX). The prerequisite for the combination to work is that concurrent solve can generate enough parallelism for the later parallel solve.

Earlier studies have showed that on average there can be many nodes waiting to be explored. One is tempted to conclude that using multiple solvers the available parallelism will multiply. The following figures show that this not always true.

Fig. 5, 6, 7, 8, 9, 10 show the amount of parallelism (number of nodes to be explored) during the search with each worker for models M02, M03, M04, M05, M06, and M07, respectively. We use 16 processes (each with a different parameter set) and 4 threads per process. In each plot different curves show the amount of nodes at different solvers. For M02, M03, M04, M05, and M06, the range is between 0 and several thousands. At any moment the number of nodes in all solver trees is certainly not 16 times that in the standalone CPLEX. Sometimes there are even fewer nodes collectively in the 16 trees than in the standalone CPLEX tree. Yet this is not bad because it shows that the pruning through shared bounds is effective. In M07, however, the amount of nodes steadily increases and

that is where we expect the combination search scheme will work.

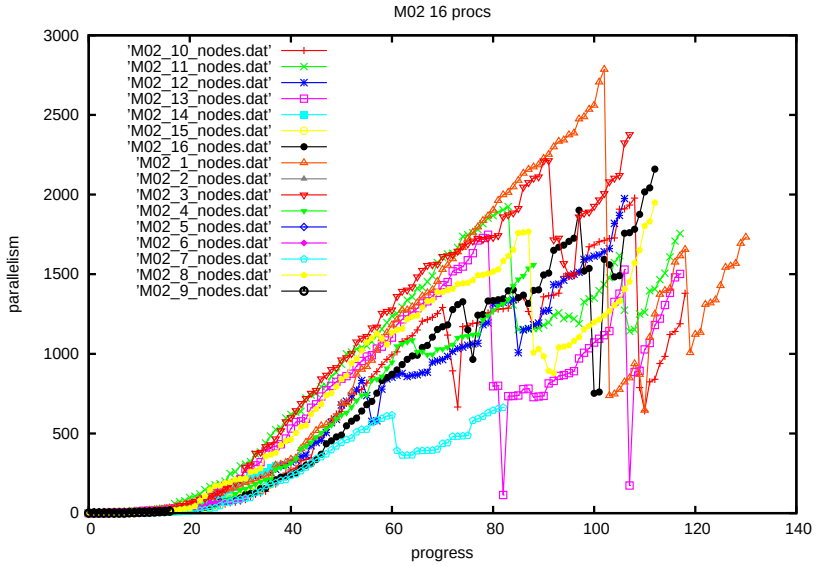

Figure 5: M02

In the ultimate, massive-scale solver for the UC models, two scenarios are most likely. Either the problem is solved completely in the concurrent solve phase, or a large number of nodes are left for the parallel solve phase. We believe the combination approach is the best candidate for utilizing modern supercomputers to solve large MIP problems.

7 Conclusion and future work

We present two versions of distributed-memory solver implemented in MPI. One adopts the coordinated, concurrent approach, and the other parallelizes the branch-and-cut search tree. Very promising performance results are achieved with the concurrent solver. In our study we show at least one set of parameter sets that is always faster than the standalone CPLEX and achieves good speedups for most of the model files. In fact, it is ready to be evaluated with many more processors. The current parallel search approach does not perform well. Although the implementation itself can be further fine tuned and improved, we do not believe that it alone is a suitable implementation for the massive scale solver.

Our study shows complementary behavior of the two approaches, and we propose a strategy for combining the two. The ultimate implementation with the advantages of the two solvers may bring drastic improvement to performance on large-scale parallel computers.

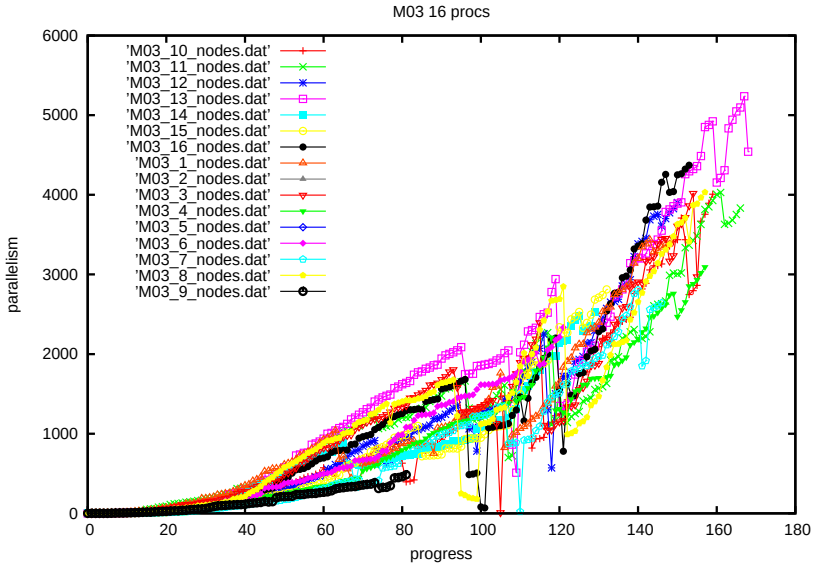

Figure 6: M03

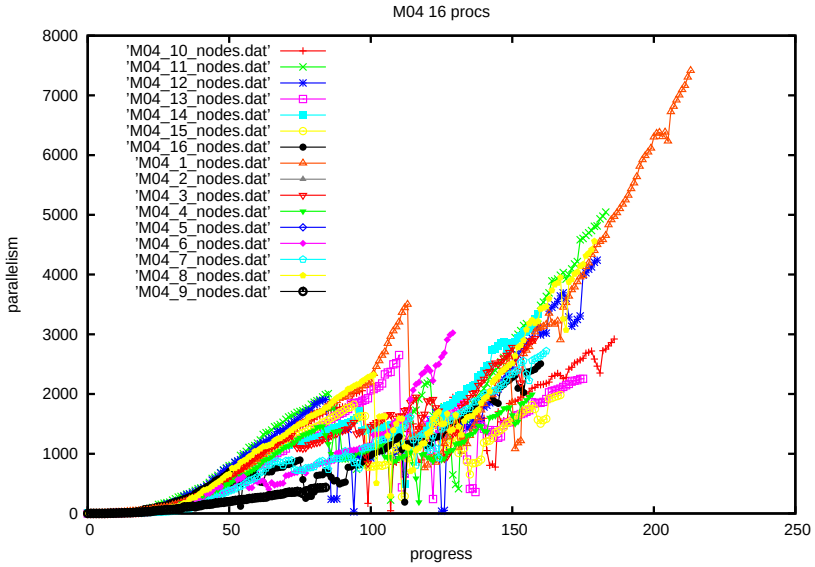

Figure 7: M04

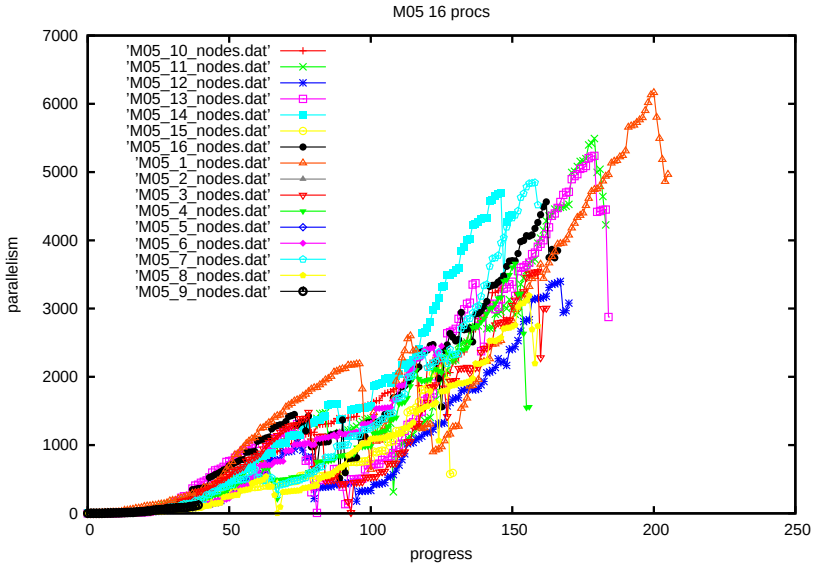

Figure 8: M05

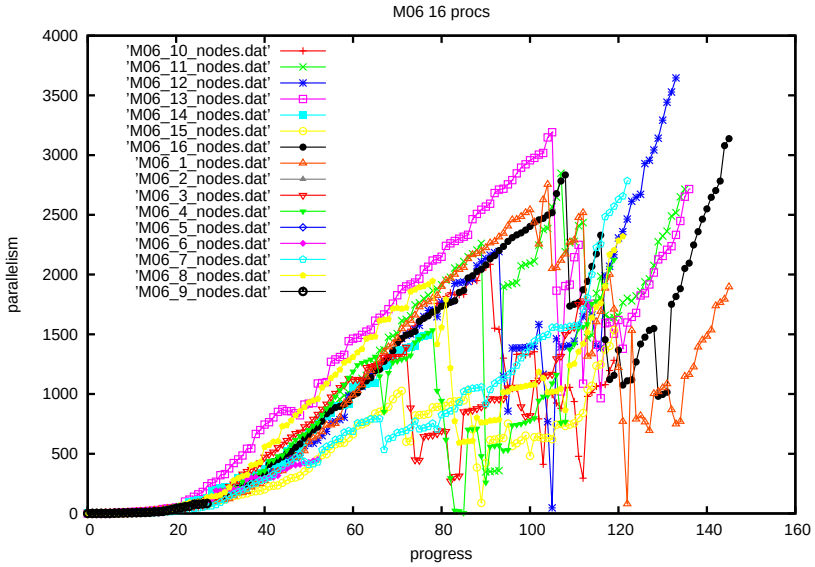

Figure 9: M06

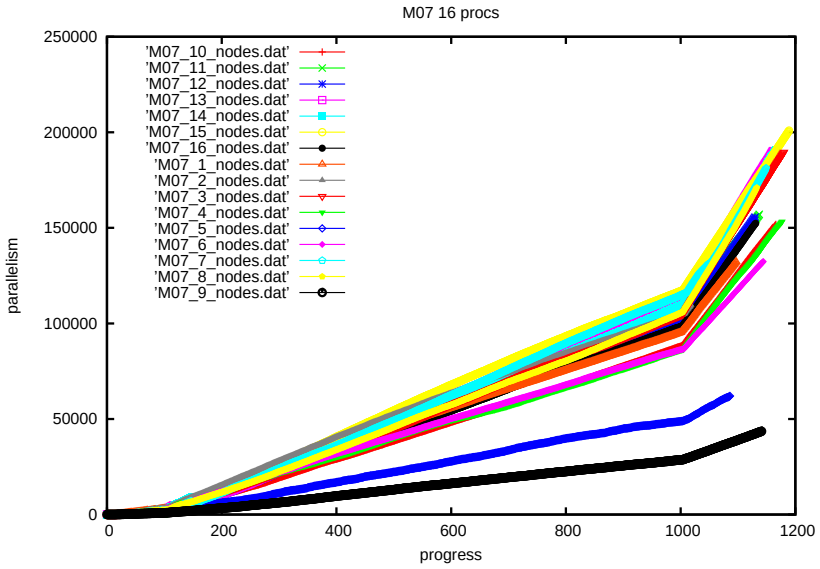

Figure 10: M07

Based on these results, IBM would like to explore a future engagement with LLNL to continue research in this important field. As part of such research, we anticipate evaluating the concurrent solve with more processes. As mentioned in Section 3.1, setting different random seeds is a way of increasing parallelism for the distributed memory implementation. We also would like to study the best approach to combine the two solvers described in this report and to conduct more extensive studies with other mixed integer programming problems.

A Parameter Sets

```
static struct paramvalue nocuts[] = {
   LONGPARAM (CPX_PARAM_CUTPASS, -1),
   ENDPARAM
};

static struct paramvalue aggrootcuts[]= {
   INTPARAM (CPX_PARAM_CLIQUES, 1),
   INTPARAM (CPX_PARAM_COVERS, 3),
   INTPARAM (CPX_PARAM_FLOWCOVERS, 1),
   INTPARAM (CPX_PARAM_FRACCUTS, 2),
   INTPARAM (CPX_PARAM_GUBCOVERS, 1),
   INTPARAM (CPX_PARAM_IMPLBD, 2),
   INTPARAM (CPX_PARAM_MIRCUTS, 1),
   INTPARAM (CPX_PARAM_MCFCUTS, 1),
   INTPARAM (CPX_PARAM_FLOWPATHS, 1),
   INTPARAM (CPX_PARAM_ZEROHALFCUTS, 1),
   ENDPARAM
};
static struct paramvalue aggprob[] = {
   INTPARAM (CPX_PARAM_PRESLVND, 2),
   INTPARAM (CPX_PARAM_PROBE, 3),
   ENDPARAM
};

static struct paramvalue nonodecuts[]= {
   INTPARAM (CPX_PARAM_COVERS, 1),
   INTPARAM (CPX_PARAM_IMPLBD, 1),
   ENDPARAM
};


static struct paramvalue allprimal[] = {
   INTPARAM (CPX_PARAM_STARTALG, 1),
   INTPARAM (CPX_PARAM_SUBALG, 1),
   ENDPARAM
};

static struct paramvalue moreins[] = {
   INTPARAM (CPX_PARAM_RINSHEUR, 100),
   ENDPARAM
};
```

```
static struct paramvalue moregomory[] = {
   INTPARAM (CPX_PARAM_FRACCAND, 10000),
   LONGPARAM (CPX_PARAM_FRACPASS, 10),
   ENDPARAM
};

static struct paramvalue moreprob[] = {
   INTPARAM (CPX_PARAM_PROBE, 1),
   ENDPARAM
};

static struct paramvalue noprob[] = {
   INTPARAM (CPX_PARAM_PROBE, -1),
   ENDPARAM
};

static struct paramvalue aggrheur[] = {
   LONGPARAM (CPX_PARAM_HEURFREQ, 3),
   INTPARAM (CPX_PARAM_RINSHEUR, 20),
   ENDPARAM
};

static struct paramvalue noheur[] = {
   LONGPARAM (CPX_PARAM_HEURFREQ, -1),
   ENDPARAM
};

static struct paramvalue moreheur[] = {
   DOUBLEPARAM (CPX_PARAM_HEUREFFORT,1e+75 ),
   ENDPARAM
};

static struct paramvalue nonodeheur[] = {
   LONGPARAM (CPX_PARAM_HEURFREQ, 100000000),
   INTPARAM (CPX_PARAM_RINSHEUR, 100000000),
   ENDPARAM
};

static struct paramvalue startprimal[] = {
   INTPARAM (CPX_PARAM_STARTALG, 1),
   ENDPARAM
};

static struct paramvalue startbarrier[] = {
```

```
   INTPARAM (CPX_PARAM_STARTALG, 4),
   ENDPARAM
};

static struct paramvalue fastnodes[] = {
   INTPARAM (CPX_PARAM_COVERS, 1),
   INTPARAM (CPX_PARAM_IMPLBD, 1),
   LONGPARAM (CPX_PARAM_HEURFREQ, 100000000),
   INTPARAM (CPX_PARAM_RINSHEUR, 100000000),
   ENDPARAM
};

static struct paramvalue bestbound[]= {
   INTPARAM (CPX_PARAM_CLIQUES, 3),
   INTPARAM (CPX_PARAM_COVERS, 3),
   INTPARAM (CPX_PARAM_FLOWCOVERS, 2),
   INTPARAM (CPX_PARAM_FRACCUTS, 2),
   INTPARAM (CPX_PARAM_GUBCOVERS, 2),
   INTPARAM (CPX_PARAM_IMPLBD, 2),
   INTPARAM (CPX_PARAM_MIRCUTS, 2),
   INTPARAM (CPX_PARAM_MCFCUTS, 2),
   INTPARAM (CPX_PARAM_FLOWPATHS, 2),
   INTPARAM (CPX_PARAM_ZEROHALFCUTS, 2),
   INTPARAM (CPX_PARAM_PRESLVND, 2),
   INTPARAM (CPX_PARAM_PROBE, 3),
   DOUBLEPARAM(CPX_PARAM_BTTOL, 0.1),
   ENDPARAM
};
static struct paramvalue steepedge[] = {
   INTPARAM (CPX_PARAM_DPRIIND, 2),
   ENDPARAM
};

static struct paramvalue purebb[] = {
   LONGPARAM (CPX_PARAM_HEURFREQ, 100000000),
   INTPARAM (CPX_PARAM_RINSHEUR, 100000000),
   INTPARAM (CPX_PARAM_CUTPASS, -1),
   ENDPARAM
};
static struct paramvalue fewcuts[] = {
   LONGPARAM (CPX_PARAM_CUTPASS, 1),
   ENDPARAM
};
```

```
static struct paramvalue aggcuts[]= {
   INTPARAM (CPX_PARAM_CLIQUES, 3),
   INTPARAM (CPX_PARAM_COVERS, 3),
   INTPARAM (CPX_PARAM_FLOWCOVERS, 2),
   INTPARAM (CPX_PARAM_FRACCUTS, 2),
   INTPARAM (CPX_PARAM_GUBCOVERS, 2),
   INTPARAM (CPX_PARAM_IMPLBD, 2),
   INTPARAM (CPX_PARAM_MIRCUTS, 2),
   INTPARAM (CPX_PARAM_MCFCUTS, 2),
   INTPARAM (CPX_PARAM_FLOWPATHS, 2),
   INTPARAM (CPX_PARAM_ZEROHALFCUTS, 2),
   ENDPARAM
};

static struct paramvalue dualbound[] = {
   LONGPARAM (CPX_PARAM_HEURFREQ, -1), /* Heuristics off. */
   /* All cuts aggressive. */
   INTPARAM (CPX_PARAM_CLIQUES, 3),
   INTPARAM (CPX_PARAM_COVERS, 3),
   INTPARAM (CPX_PARAM_DISJCUTS, 3),
   INTPARAM (CPX_PARAM_FLOWCOVERS, 2),
   INTPARAM (CPX_PARAM_FRACCUTS, 2),
   INTPARAM (CPX_PARAM_GUBCOVERS, 2),
   INTPARAM (CPX_PARAM_IMPLBD, 2),
   INTPARAM (CPX_PARAM_MIRCUTS, 2),
   INTPARAM (CPX_PARAM_ZEROHALFCUTS, 2),
   INTPARAM (CPX_PARAM_MCFCUTS, 2),
   ENDPARAM
};


static struct paramvalue defaultbound[]= {
   ENDPARAM
};
```